

6

Bitcoin Architecture

Bitcoin was the first application of blockchain technology and has started a revolution with the introduction of the very first fully decentralized digital currency. It has proven to be remarkably secure and valuable as a digital currency, despite being highly volatile. The invention of Bitcoin has also sparked great interest within academia and industry and opened many new research areas. In this chapter, we shall introduce Bitcoin in detail.

Specifically, in this chapter, we will focus our attention upon the fundamentals of Bitcoin, how transactions are constructed and used, transaction structures, addresses, accounts, and mining, over the following topic areas:

- Introducing Bitcoin
- Cryptographic keys
- Addresses
- Transactions
- Blockchain
- Miners
- Network
- Wallets

Let's begin with an overview of Bitcoin.

Introducing Bitcoin

Since its introduction in 2008 by Satoshi Nakamoto, **Bitcoin** has gained immense popularity and is currently the most successful digital currency in the world, with billions of dollars invested in it.

Its popularity is also evident from the high number of users and investors, daily news related to Bitcoin, and the many start-ups and companies that are offering Bitcoin-based online exchanges. It is now also traded as Bitcoin futures on the **Chicago Mercantile Exchange (CME)**.



Interested readers can read more about Bitcoin futures at <http://www.cmegroup.com/trading/bitcoin-futures.html>.

In 2008, Bitcoin was introduced in a paper called *Bitcoin: A Peer-to-Peer Electronic Cash System*. This paper is available at <https://bitcoin.org/en/bitcoin-paper>. The name of the author, Satoshi Nakamoto, is believed to be a pseudonym, as the true identity of the inventor of Bitcoin is unknown and is the subject of much speculation.

The first key idea introduced in the paper was of purely P2P electronic cash that does not need an intermediary bank to transfer payments between peers. However, Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform. It is a combination of a P2P network, protocols, and software that facilitates the creation and usage of the digital currency. Nodes in this P2P network talk to each other using the Bitcoin protocol.

Bitcoin is built on decades of research. Various ideas and techniques from cryptography and distributed computing such as Merkle trees, hash functions, and digital signatures were used to design Bitcoin. Other ideas such as BitGold, b-money, hashcash, and cryptographic time-stamping also provided some groundwork for the invention of Bitcoin. Ideas from many of these developments were ingeniously used in Bitcoin to create the first ever truly decentralized currency. Bitcoin solves several historically difficult problems related to electronic cash and distributed systems, including:

- The Byzantine generals problem
- Sybil attacks
- The double-spending problem

The double-spending problem arises when, for example, a user sends coins to two different users at the same time, and they are verified independently as valid transactions. The double-spending problem is resolved in Bitcoin by using a distributed ledger (the blockchain) where every transaction is recorded permanently, and by implementing a transaction validation and confirmation mechanism.

In this chapter, we will look at the various actors and components of the Bitcoin network, and how they interact to form it:

- Cryptographic keys
- Addresses
- Transactions
- Blockchain
- Miners
- Network
- Wallets

First, we will look at the keys and addresses that are used to represent the ownership and transfer of value on the Bitcoin network.

Cryptographic keys

On the Bitcoin network, possession of bitcoins and the transfer of value via transactions are reliant upon private keys, public keys, and addresses. **Elliptic Curve Cryptography (ECC)** is used to generate public and private key pairs in the Bitcoin network. We have already covered these concepts in *Chapter 4, Asymmetric Cryptography*, and here we will see how private and public keys are used in the Bitcoin network.

Private keys in Bitcoin

Private keys are required to be kept safe and normally reside only on the owner's side. Private keys are used to digitally sign transactions, proving ownership of bitcoins.

Private keys are fundamentally 256-bit numbers randomly chosen in the range specified by the SECP256K1 ECDSA curve recommendation. Any randomly chosen 256-bit number from 0x1 to 0xFFFF FFFF FFFF FFFF FFFF FFFF BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 is a valid private key.

Private keys are usually encoded using **Wallet Import Format (WIF)** in order to make them easier to copy and use. It is a way to represent the full-size private key in a different format. WIF can be converted into a private key and vice versa. For example, consider the following private key:

```
A3ED7EC8A03667180D01FB4251A546C2B9F2FE33507C68B7D9D4E1FA5714195201
```

When converted into WIF format, it looks as shown here:

```
L2iN7umV7kbr6LuCmgM27rBnptGbDVc8g4ZBm6EbgTPQXnj1RCZP
```



Interested readers can do some experimentation using the online tool available at <http://gobittest.appspot.com/PrivateKey>.

Also, **mini private key format** is sometimes used to create a private key with a maximum of 30 characters to allow storage where physical space is limited. For example, etching on physical coins or encoding in damage resistant QR codes. The QR code is more damage resistant because more dots can be used for error correction and fewer for encoding the private key.



QR codes use **Reed-Solomon error correction**. Discussion on the error correction mechanism and its underlying details is out of the scope of this book but readers are encouraged to research QR code error correction if interested.

A private key encoded using mini private key format is also sometimes called a **minikey**. The first character of the mini private key is always the uppercase letter S. A mini private key can be converted into a normal-sized private key, but an existing normal-sized private key cannot be converted into a mini private key. This format was used in **Casascius** physical bitcoins. The Bitcoin core client also allows the encryption of the wallet that contains the private keys.

As we learned in *Chapter 4, Asymmetric Cryptography*, private keys have their own corresponding public keys. Public or private keys on their own are useless—pairs of public and private keys are required for the normal functioning of any public key cryptography-based systems such as the Bitcoin blockchain.

Public keys in Bitcoin

All network participants can see public keys on the blockchain. Public keys are derived from private keys due to their special mathematical relationship. Once a transaction signed with the private key is broadcast on the Bitcoin network, public keys are used by the nodes to verify that the transaction has indeed been signed with the corresponding private key. This process of verification proves the ownership of the Bitcoin.

Bitcoin uses ECC based on the SECP256K1 standard. More specifically, it makes use of an **Elliptic Curve Digital Signature Algorithm (ECDSA)** to ensure that funds remain secure and can only be spent by the legitimate owner. If you need to refresh the relevant cryptography concepts, you can refer to *Chapter 4, Asymmetric Cryptography*, where ECC was explained. Public keys can be represented in uncompressed or compressed format and are fundamentally x and y coordinates on an elliptic curve. The compressed version of public keys includes only the x part since the y part can be derived from it.

The reason why the compressed version of public keys works is that if the ECC graph is visualized, it reveals that the y coordinate can be either below the x axis or above the x axis, and as the curve is symmetric, only the location in the prime field is required to be stored. If y is even, then its value lies above the x axis, and if it is odd, then it is below the x axis. This means that instead of storing both x and y as the public key, only x needs to be stored with the information about whether y is even or odd.

Initially, the Bitcoin client used uncompressed keys, but starting from Bitcoin Core client 0.6, compressed keys are used as standard. This resulted in an almost 50% reduction of space used to store public keys in the blockchain.

Keys are identified by various prefixes, described as follows:

- Uncompressed public keys use `0x04` as the prefix. Uncompressed public keys are 65 bytes long. They are encoded as 256-bit unsigned big-endian integers (32 bytes), which are concatenated together and finally prefixed with a byte `0x04`. This means 1 byte for the `0x04` prefix, 32 bytes for the x integer, and 32 bytes for the y integer, which makes it 65 bytes in total.
- Compressed public keys start with `0x03` if the y 32-byte (256-bit) part of the public key is odd. It is 33 bytes in length as 1 byte is used by the `0x03` prefix (depicting an odd y) and 32 bytes are used for storing the x coordinate.
- Compressed public keys start with `0x02` if the y 32-byte (256-bit) part of the public key is even. It is 33 bytes in length as 1 byte is used by the `0x02` prefix (depicting an even y) and 32 bytes are used for storing the x coordinate.

Having talked about private and public keys, let's now move on to another important aspect of Bitcoin: addresses derived from public keys.

Addresses

The following diagram shows how an address is generated, from generating the private key to the final output of the Bitcoin address:

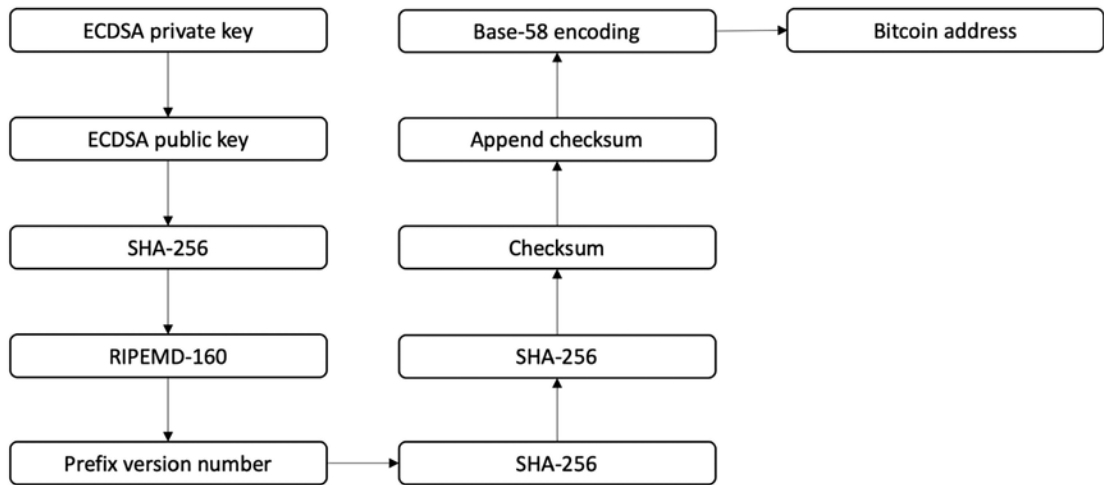


Figure 6.1: Address generation in Bitcoin

In the preceding diagram, there are a number of steps:

1. In the first step, we have a randomly generated ECDSA private key.
2. The public key is derived from the ECDSA private key.
3. The public key is hashed using the SHA-256 cryptographic hash function.
4. The hash generated in *step 3* is hashed using the RIPEMD-160 hash function.
5. The version number is prefixed to the RIPEMD-160 hash generated in *step 4*.
6. The result produced in *step 5* is hashed using the SHA-256 cryptographic hash function.
7. SHA-256 is applied again.
8. The first 4 bytes of the result produced from *step 7* are the address checksum.
9. This checksum is appended to the RIPEMD-160 hash generated in *step 4*.
10. The resultant byte string is encoded into a Base58-encoded string by applying the Base58 encoding function.
11. Finally, the result is a typical Bitcoin address.

Typical Bitcoin addresses

Bitcoin addresses are 26-35 characters long and begin with the digits 1 or 3. A typical Bitcoin address looks like the string shown here:

```
15ccPQG3PQXcj7fhgmWAHN7SQ7JBvfNFGb
```

Addresses are also commonly encoded in a QR code for easy distribution. The QR code of the preceding Bitcoin address is shown in the following image:



Figure 6.2: QR code of the Bitcoin address 15ccPQG3PQXcj7fhgmWAHN7SQ7JBvfNFGb

Currently, there are two types of addresses, the commonly used P2PKH and the P2SH type (both defined later in this chapter), starting with numbers 1 and 3, respectively. In the early days, Bitcoin used direct Pay-to-Pubkey, which has now been superseded by P2PKH. These types will be explained later in the chapter. However, direct Pay-to-Pubkey is still used in Bitcoin for coinbase addresses. Addresses should not be used more than once; otherwise, privacy and security issues can arise.

Avoiding address reuse circumvents anonymity issues to an extent, but Bitcoin has some other security issues as well, such as transaction malleability, Sybil attacks, race attacks, and selfish mining, all of which require different approaches to resolve. Transaction malleability has been resolved with the so-called SegWit soft-fork upgrade of the Bitcoin protocol. This concept will be explained later in the chapter.



Figure 6.3: From bitaddress.org, a private key and Bitcoin address in a paper wallet

Bitcoin addresses are encoded using the Base58Check encoding. This encoding is used to limit the confusion between various characters, such as 0/O or I/l, as they can look the same in different fonts. The encoding basically takes the binary byte arrays and converts them into human-readable strings. This string is composed by utilizing a set of 58 alphanumeric symbols. More explanation and logic can be found in the `base58.h` source file in the Bitcoin source code:

```
/**
 * Why base-58 instead of standard base-64 encoding?
 * - Don't want 00IL characters that look the same in some fonts and
 *   could be used to create visually identical looking data.
 * - A string with non-alphanumeric characters is not as easily accepted as input.
 * - E-mail usually won't line-break if there's no punctuation to break at.
 * - Double-clicking selects the whole string as one word if it's all alphanumeric.
 */
```



This file is present in the Bitcoin source code and can be viewed at <https://github.com/bitcoin/bitcoin/blob/c8971547d9c9460fcbec6f54888df83f002c3dfd/src/base58.h>.

Advanced Bitcoin addresses

In addition to common types of addresses in Bitcoin, there are some advanced types of addresses available in Bitcoin too:

- **Vanity addresses:** As Bitcoin addresses are based on Base58 encoding, it is possible to generate addresses that contain human-readable messages and are personalized. An example is `1BasHiry2VoCQcdX6X64oxvKRuf7fW6qGr`—note that the address contains the name `BasHir`. Vanity addresses are generated using a brute-force method. There are various online services that provide this service. More details on the mechanism and a link to a standalone command line program is available at <https://en.bitcoin.it/wiki/Vanitygen>.
- **Multi-signature addresses:** As the name implies, these addresses require multiple private keys. In practical terms, this means that in order to release the coins, a certain set number of signatures is required. This is also known as *M of N multisig*. Here, *M* represents the threshold or minimum number of signatures required from *N* number of keys to release the bitcoins. Remember that we discussed this concept in *Chapter 4, Asymmetric Cryptography*.

With this section, we've finished our introduction to addresses in Bitcoin. In the next section, we will introduce Bitcoin transactions, which are the most fundamental and important aspect of Bitcoin.

Transactions

Transactions are at the core of the Bitcoin ecosystem. Transactions can be as simple as just sending some bitcoins to a Bitcoin address, or can be quite complex, depending on the requirements.

A Bitcoin transaction is composed of several elements:

- **Transaction ID:** A 32 byte long unique transaction identifier
- **Size:** This is the size of the transaction in bytes.
- **Weight:** This is a metric given for the block and transaction sizes since the introduction of the SegWit soft-fork version of Bitcoin.
- **Time:** This is the time when the block containing this transaction was mined.
- **Included in block:** This shows the block number on the blockchain in which the transaction is included.
- **Confirmations:** This is the number of confirmations completed by miners for this transaction.
- **Total input:** This is the number of total inputs in the transaction.
- **Total output:** This is the number of total outputs from the transaction.
- **Fees:** This is the total fee charged.
- **Fee per byte:** This field represents the total fee divided by the number of bytes in the transaction; for example, 10 Satoshis per byte.
- **Fee per weight unit:** For legacy transactions, this is calculated using the total number of bytes * 4. For SegWit transactions, it is calculated by combining a SegWit marker, flag, and witness field as one weight unit, and each byte of the other fields as four weight units.
- **Input Index:** This is the sequence number of the input.
- **Output index:** This is the sequence number of the output.
- **Output address:** This is where the bitcoins are going to.
- **Previous transaction ID:** This is the transaction ID of the previous transaction whose output(s) is used as input(s) in this transaction.
- **Previous output index:** This is the index of the previous output showing which output has been used as input in this transaction.
- **Value:** This is the amount of bitcoins.
- **Input address:** This is the address where the input is from.
- **Pkscript:** This is the unlock script for the input(s).
- **SigScript:** This is the signature for unlocking the input.
- **Witness:** This is the witness for this transaction—used only in SegWit.

Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction, and outputs as coins being created. If a transaction is minting (mining) new coins rather than spending previously created coins, then there is no input, and therefore no signature is needed. This transaction is called a coinbase transaction.

Coinbase transactions

A coinbase transaction or generation transaction is always created by a miner and is the first transaction in a block. It is used to create new coins. It includes a special field, also called the **coinbase**, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data storage.

A coinbase transaction input has the same number of fields as a usual transaction input, but the structure contains the coinbase data size and fields instead of the unlocking script size and fields. Also, it does not have a reference pointer to the previous transaction. This structure is shown in the following table:

| Field | Size | Description |
|----------------------|-----------|--|
| Transaction hash | 32 bytes | Set to all zeroes as no hash reference is used |
| Output index | 4 bytes | Set to 0xFFFFFFFF |
| Coinbase data length | 1-9 bytes | 2-100 bytes |
| Data | Variable | Any data |
| Sequence number | 4 bytes | Set to 0xFFFFFFFF |

On the other hand, if a transaction should send coins to some other user (a Bitcoin address), then it needs to be signed by the sender with their private key. In this case, a reference is also required to the previous transaction to show the origin of the coins. Coins are unspent transaction outputs represented in Satoshis.



Bitcoin, being a digital currency, has various denominations. The smallest Bitcoin denomination is the **Satoshi**, which is equivalent to 0.00000001 BTC.

The transaction lifecycle

Now, let's look at the lifecycle of a Bitcoin **transaction**. The steps of the process are as follows:

1. A user/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcast to the Bitcoin network using a flooding algorithm, which is an algorithm to distribute data to every node in the network.
4. Mining nodes (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transactions are placed in the block, they are placed in a special memory buffer called the transaction pool.
5. Next, the mining starts, which is the process through which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources. Once a miner solves the **Proof of Work (PoW)** problem, it broadcasts the newly mined block to the network. The nodes verify the block and propagate the block further, and confirmations start to generate.

6. Finally, the confirmations start to appear in the receiver's wallet and after approximately three confirmations, the transaction is considered finalized and confirmed. However, three to six is just the recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after six confirmations.

When a transaction is created by a user and sent to the network, it ends up in a special area on each Bitcoin software client. This special area is called the transaction pool. Also known as memory pools, transaction pools are created in local memory (computer RAM) by nodes (Bitcoin clients) to maintain a temporary list of transactions that have not yet been added to a block. Miners pick up transactions from these memory pools to create candidate blocks. Miners select transactions from the pool after they pass the verification and validity checks. We introduce how Bitcoin transactions are validated in the next section.

Transaction validation

This verification process is performed by Bitcoin nodes. There are three main things that nodes check when verifying a transaction:

1. That transaction inputs are previously unspent. This validation step prevents double spending by verifying that the transaction inputs have not already been spent by someone else.
2. That the sum of the transaction outputs is not more than the total sum of the transaction inputs. However, both input and output sums can be the same, or the sum of the input (total value) could be more than the total value of the outputs. This check ensures that no new bitcoins are created out of thin air.
3. That the digital signatures are valid, which ensures that the script is valid.

To send transactions on the Bitcoin network, the sender needs to pay a fee to the miners. The selection of which transactions to choose is based on the fee and their place in the order of transactions in the pool. Miners prefer to pick up transactions with higher fees.

Transaction fees

Transaction fees are charged by the miners. The fee charged is dependent upon the size and weight of the transaction. Transaction fees are calculated by subtracting the sum of the inputs from the sum of the outputs:

$$fee = sum(inputs) - sum(outputs)$$

The fees are used as an incentive for miners to encourage them to include users' transactions in the block the miners are creating. All transactions end up in the memory pool, from which miners pick up transactions based on their priority to include them in the proposed block. The calculation of priority is introduced later in this chapter; however, from a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners. There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes.

Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course but may take a very long time. This is, however, no longer practical due to the high volume of transactions and competing investors on the Bitcoin network; therefore, it is advisable to always provide a fee. The time taken for transaction confirmation usually ranges from 10 minutes to over 12 hours in some cases. The transaction time is also dependent on network activity. If the network is very busy, then naturally, transactions will take longer to process.

At times in the past, Bitcoin fees were so high that even for smaller transactions, a high fee was charged. This was due to the fact that miners are free to choose which transactions they pick to verify and add to a block, and they naturally select the ones with higher fees. A high number of users creating thousands of transactions also played a role in causing this situation of high fees because transactions were competing to be picked up first and miners picked up the ones with the highest fees. This fee is also usually estimated and calculated by the Bitcoin wallet software automatically before sending the transaction.

Mining and miners are concepts that we will look at a bit later in this chapter in the section about *Mining*.

The transaction data structure

A transaction, at a high level, contains metadata, inputs, and outputs. Transactions are combined to create a block's body. The general transaction data structure is shown in the following table:

| Field | Size | Description |
|-------------------|---------------------------|---|
| Version number | 4 bytes | Specifies the rules to be used by the miners and nodes for transaction processing. There are two versions of transactions, that is, 1 and 2. |
| Flag | 2 bytes or none | Either always 0001 or none, used to indicate the presence of witness data. |
| Input counter | 1–9 bytes | The number (a positive integer) of inputs included in the transaction. |
| List of inputs | Variable | Each input is composed of several fields. These include: <ul style="list-style-type: none"> • The previous transaction hash • The index of the previous transaction • Transaction script length • Transaction script • Sequence number |
| Output counter | 1–9 bytes | A positive integer representing the number of outputs. |
| List of outputs | Variable | Outputs included in the transaction. This field depicts the target recipient(s) of the bitcoins. |
| List of witnesses | Based on how many or none | Witnesses—1 for each input. Absent if the <i>Flag</i> field is absent. |

| | | |
|-----------|---------|---|
| Lock time | 4 bytes | This field defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or the block height. |
|-----------|---------|---|

The following is a sample decoded transaction:

```
{
  "txid": "d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d",
  "hash": "d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d",
  "version": 1,
  "size": 226,
  "vsize": 226,
  "weight": 904,
  "locktime": 0,
  "vin": [
    {
      "txid": "40120e43f00ff96e098a9173f14f1371655b3478bc0a558d6dc17a4ab176387d",
      "vout": 139,
      "scriptSig": {
        "asm":
"3045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7afb92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c71758d[ALL]02c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12",
        "hex":
"483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7afb92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c71758d012102c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00033324,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160
c568ffeb46c6a9362e44a5a49deaa6eab05a619a OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c568ffeb46c6a9362e44a5a49deaa6eab05a619a88ac",
        "address": "1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3",
        "type": "pubkeyhash"
      }
    }
  ]
}
```

```
    },
    {
      "value": 0.00093376,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160
9386c8c880488e80a6ce8f186f788f3585f74aee OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a9149386c8c880488e80a6ce8f186f788f3585f74aee88ac",
        "address": "1ET3oBGf8JpunjytE7owyVtmBjmvcDycQe",
        "type": "pubkeyhash"
      }
    }
  ]
}
```

As shown in the preceding decoded transaction, there are several structures that make up a transaction. All these elements will be described now.

Metadata

This part of the transaction contains values such as the size of the transaction, the number of inputs and outputs, the hash of the transaction, and a locktime field. Every transaction has a prefix specifying the version number. These fields are shown in the preceding example as locktime, size, weight, and version.

Inputs

Generally, each input (vin) spends a previous output. Each output is considered an **Unspent Transaction Output (UTXO)** until an input consumes it. A UTXO can be spent as an input to a new transaction. The transaction input data structure is explained in the following table:

| Field | Size | Description |
|------------------|-----------|---|
| Transaction hash | 32 bytes | The hash of the previous transaction with UTXO |
| Output index | 4 bytes | This is the previous transaction's output index, such as UTXO to be spent |
| Script length | 1-9 bytes | The size of the unlocking script |
| Unlocking script | Variable | The input script (<code>ScriptSig</code>), which satisfies the requirements of the locking script |
| Sequence number | 4 bytes | Usually disabled or contains lock time— it being disabled is represented by <code>0xFFFFFFFF</code> |

In the previous sample decoded transaction, the inputs are defined under the "inputs" : [section.

Outputs

Outputs (vout) have three fields, and they contain instructions for sending bitcoins. The first field contains the amount of Satoshis, whereas the second field contains the size of the locking script. Finally, the third field contains a locking script that holds the conditions that need to be met for the output to be spent. More information on transaction spending using locking and unlocking scripts and producing outputs is discussed later in this section.

The transaction output data structure is explained in the following table:

| Field | Size | Description |
|----------------|-----------|---|
| Value | 8 bytes | The total number (in positive integers) of Satoshis to be transferred |
| Script size | 1-9 bytes | Size of the locking script |
| Locking script | Variable | Output script (ScriptPubKey) |

In the previous sample decoded transaction, two outputs are shown under the "vout":[section.

Verification

Verification is performed using Bitcoin's scripting language where transactions' cryptographic signatures are checked for validity, all inputs and outputs are checked, and the sum of all inputs must be equal to or greater than the sum of all outputs.

Now that we've covered the transaction lifecycle and data structure, let's move on to talk about the scripts used to undertake these transactions.

The Script language

Bitcoin uses a simple stack-based language called **Script** to describe how bitcoins can be spent and transferred. It is not Turing-complete and has no loops to avoid any undesirable effects of long-running/hung scripts on the Bitcoin network. This scripting language is based on a Forth programming language-like syntax and uses a reverse polish notation in which every operand is followed by its operators. It is evaluated from left to right using a **Last in, First Out (LIFO)** stack.

Scripts are composed of two components, namely elements and operations. Scripts use various operations (**opcodes**) or instructions to define their operations. Elements simply represent data such as digital signatures. Opcodes are also known as words, commands, or functions. Earlier versions of the Bitcoin node software had a few opcodes that are no longer used due to bugs discovered in their design.

The various categories of scripting opcodes are constants, flow control, stack, bitwise logic, splice, arithmetic, cryptography, and lock time.

A transaction script is evaluated by combining ScriptSig and ScriptPubKey:

- ScriptSig is the unlocking script, provided by the user who wishes to unlock the transaction

- `ScriptPubKey` is the locking script, is part of the transaction output, and specifies the conditions that need to be fulfilled in order to unlock and spend the output

We will look at a script execution in detail shortly.

Opcodes

In a computer, an opcode is an instruction to perform some operation. For example, `ADD` is an opcode, which is used for integer addition in Intel CPUs and various other architectures. Similarly, in Bitcoin design, opcodes are introduced that perform several operations related to Bitcoin transaction verification.

Descriptions of some of the most commonly used opcodes are listed in the following table, extracted from the Bitcoin Developer's Guide:

| Opcode | Description |
|-------------------------------|---|
| <code>OP_CHECKSIG</code> | This takes a public key and signature and validates the signature of the hash of the transaction. If it matches, then <code>TRUE</code> is pushed onto the stack; otherwise, <code>FALSE</code> is pushed. |
| <code>OP_EQUAL</code> | This returns 1 if the inputs are exactly equal; otherwise, 0 is returned. |
| <code>OP_DUP</code> | This duplicates the top item in the stack. |
| <code>OP_HASH160</code> | The input is hashed twice, first with SHA-256 and then with RIPEMD-160. |
| <code>OP_VERIFY</code> | This marks the transaction as invalid if the top stack value is not true. |
| <code>OP_EQUALVERIFY</code> | This is the same as <code>OP_EQUAL</code> , but it runs <code>OP_VERIFY</code> afterward. |
| <code>OP_CHECKMULTISIG</code> | This instruction takes the first signature and compares it against each public key until a match is found and repeats this process until all signatures are checked. If all signatures turn out to be valid, then a value of 1 is returned as a result; otherwise, 0 is returned. |
| <code>OP_HASH256</code> | The input is hashed twice with SHA-256. |
| <code>OP_MAX</code> | This returns the larger value of two inputs. |

There are many opcodes in the Bitcoin scripting language and covering all of them is out of the scope of this book. However, all opcodes are declared in the `script.h` file in the Bitcoin reference client source code, available at <https://github.com/Bitcoin/Bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/script/script.h#L53>.

There are various standard scripts available in Bitcoin to handle the verification and value transfer from the source to the destination. These scripts range from very simple to quite complex depending upon the requirements of the transaction.

Standard transaction scripts

Standard transactions are evaluated using the `IsStandard()` and `IsStandardTx()` tests and only those transactions that pass the tests are allowed to be broadcast or mined on the Bitcoin network.

However, nonstandard transactions are also allowed on the network, as long as they pass the validity checks:

- **Pay-to-Public-Key Hash (P2PKH):** P2PKH is the most commonly used transaction type and is used to send transactions to Bitcoin addresses. The format of this type of transaction is shown as follows:

```
ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
ScriptSig: <sig> <pubKey>
```

The ScriptPubKey and ScriptSig parameters are concatenated together and executed. An example will follow shortly in this section, where this is explained in more detail.

- **Pay-to-Script Hash (P2SH):** P2SH is used in order to send transactions to a script hash (that is, addresses starting with 3) and was standardized in BIP16. In addition to passing the script, the redeem script is also evaluated and must be valid. The template is shown as follows:

```
ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL
ScriptSig: [<sig>...<sign>] <redeemScript>
```

- **MultiSig (Pay to MultiSig):** The M of N multisignature transaction script is a complex type of script where it is possible to construct a script that requires multiple signatures to be valid in order to redeem a transaction. Various complex transactions such as escrow and deposits can be built using this script. The template is shown here:

```
ScriptPubKey: <m> <pubKey> [<pubKey> . . . ] <n> OP_CHECKMULTISIG
ScriptSig: 0 [<sig> . . . <sign>]
```

Raw multisig is obsolete, and multisig is now usually part of the P2SH redeem script, mentioned in the previous bullet point.

- **Null data/OP_RETURN:** This script is used to store arbitrary data on the blockchain for a fee. The limit of the message is 40 bytes. The output of this script is unredemable because OP_RETURN will fail the validation in any case. ScriptSig is not required in this case. The template is very simple and is shown as follows:

```
OP_RETURN <data>
```

The P2PKH script execution is shown in the following diagram:

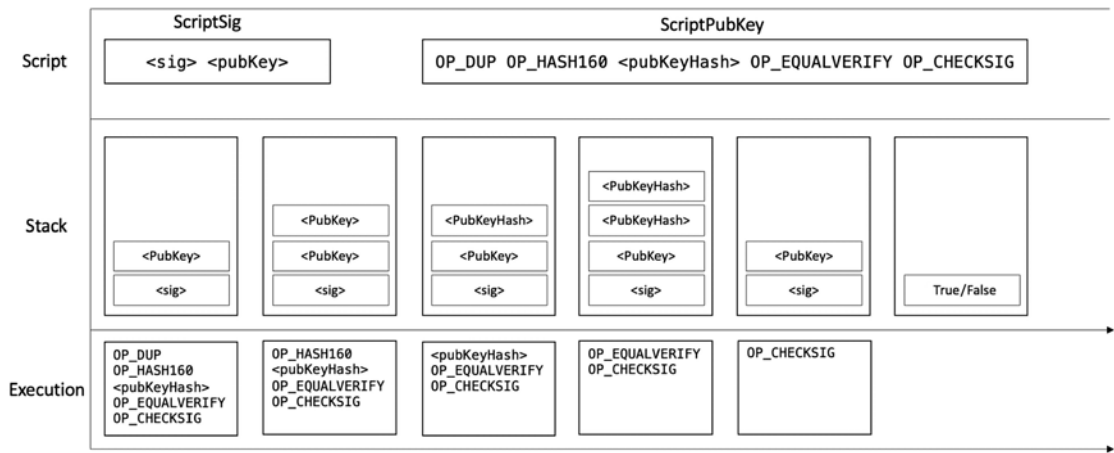


Figure 6.4: P2PKH script execution

In the preceding diagram, we have a standard P2PKH script presented at the top, which shows both the unlocking (ScriptSig) and locking (ScriptPubKey) parts of the script.

The unlocking script is composed of <sig> and <pubkey> elements, which are part of all transaction inputs. The unlocking script satisfies the conditions that are required to consume the output. The locking script defines the conditions that are required to be met to spend the bitcoins. Transactions are authorized by executing both parts collectively.

Now focusing again on Figure 6.4, we see that in the middle we have a visualization of the stack where the data elements are pushed and popped from. In the bottom section, we show the execution of the script. This diagram shows the step-by-step execution of the script with its outcome on the stack.

Now let's examine how this script is executed:

1. In the first step of the data elements, <sig> and <pubkey> are placed on the stack.
2. The stack item at the top, <pubkey>, is duplicated due to the OP_DUP instruction, which duplicates the top stack item.
3. After this, the instruction OP_HASH160 executes, which produces the hash of <pubkey>, which is the top element in the stack.
4. <pubkeyhash> is then pushed on to the stack. At this stage, we have two hashes on the stack: the one that is produced as a result of executing OP_HASH160 on <pubkey> from the unlocking script, and the other one provided by the locking script.
5. Now the OP_EQUALVERIFY opcode instruction executes and checks whether the top two elements (that is, the hashes) are equal or not. If they are equal, the script continues; otherwise, it fails.
6. Finally, OP_CHECKSIG executes to check the validity of the signatures of the top two elements of the stack. If the signature is valid then the stack will contain the value True, that is, 1; otherwise, False, that is, 0.

All transactions are encoded into hex format before being transmitted over the Bitcoin network. A sample transaction can be retrieved using `bitcoin-cli` on the Bitcoin node running on the main net as follows:

```
$ bitcoin-cli getrawtransaction
"d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d"
```

The output is shown in hex format:

```
{
  "result":
    "01000000017d3876b14a7ac16d8d550abc78345b6571134ff173918a096ef90ff043
    0e12408b0000006b483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01
    757c8e4fa7afbf92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5
    ef971e3f98ccba2c71758d012102c16942555f5e633645895c9affcb994ea7910097b
    7734a6c2d25468622f25e12fffffffff022c820000000000001976a914c568ffeb46c6
    a9362e44a5a49deaa6eab05a619a88acc06c0100000000001976a9149386c8c880488
    e80a6ce8f186f788f3585f74aee88ac00000000", "error": null, "id": null
}
```



As an alternative to client software that is locally installed, an online service can also be used, which is available here: <https://chainquery.com/bitcoin-cli/getrawtransaction>

Scripting is quite limited and can only be used to program one thing—the transfer of bitcoins from one address to other addresses. However, there is some flexibility possible when creating these scripts, which allows for certain conditions to be put on the spending of the bitcoins. This set of conditions can be considered a basic form of a financial contract.

Contracts

Contracts are Bitcoin scripts that use the Bitcoin blockchain to enforce a financial agreement. This is a simple definition but has far-reaching consequences as it allows users to programmatically create complex contracts that can be used in many real-world scenarios. Contracts allow the development of fully decentralized, independent, and reduced-risk platforms by programmatically enforcing different conditions for unlocking bitcoins. With the security guarantees provided by the Bitcoin blockchain, it is almost impossible to circumvent these conditions.



Note that these are not the same as **smart contracts**, which allow the writing of arbitrary programs on the blockchain. We will discuss these further in *Chapter 8, Smart Contracts*.

Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the Bitcoin scripting language. The current implementation of the Script language is minimal, but it is still possible to develop various types of complex contracts. For example, we could set the release of funds to be allowed only when multiple parties sign the transaction or release the funds only after a specific amount of time has elapsed. Both scenarios can be realized using the `multisig` and `transaction lock time` options.

Even though the scripting language in Bitcoin can be used to create complex contracts, it is quite limited and is not at all on par with smart contracts, which are Turing-complete constructs and allow arbitrary program development. Recently, however, there has been some more advancement in this area. A new smart contract language for Bitcoin has been announced, called **Miniscript**, which allows for a more structured approach to writing Bitcoin scripts. Even though Bitcoin Script supports combinations of different spending conditions, such as time and hash locks, it is not easy to analyze existing scripts or build new complex scripts. Miniscript makes it easier to write complex spending rules. It also makes it easier to ascertain the correctness of the script. Currently, Miniscript supports P2WSH and P2SH-P2WSH, and offers limited support for P2SH.



More information on Miniscript is available at <http://bitcoin.sipa.be/miniscript/>.

We will now introduce some of Bitcoin's infamous shortcomings.

Transaction bugs

Even though transaction construction and validation are generally secure and sound processes, some vulnerabilities exist in Bitcoin. The following are two major Bitcoin vulnerabilities that have been infamously exploited:

- **Transaction malleability** is a Bitcoin attack that was introduced due to a bug in the Bitcoin implementation. Due to this bug, it became possible for an adversary to change the transaction ID of a transaction, thus resulting in a scenario where it appears that a certain transaction has not been executed. This can allow scenarios where double deposits or withdrawals can occur. This was fixed under BIP62, which fixed several causes of malleability. You can find more details here: <https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki>.
- **Value overflow**: This incident is one of the most well-known events in Bitcoin history. On August 15, 2010, a transaction was discovered that created roughly 184 billion bitcoins. This problem occurred due to the integer overflow bug where the amount field in the Bitcoin code was defined as a signed integer instead of an unsigned integer. This bug meant that the amount could also be negative and resulted in a situation where the outputs were so large that the total value resulted in an overflow. To the validation logic in the Bitcoin code, all appeared to be correct, and it looked like the fee was also positive (after the overflow). This bug was fixed via a soft fork (more on this in the *Blockchain* section) quickly after its discovery.



More information on this vulnerability is available at <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-5139>.

Security research in general, and specifically in the world of Bitcoin (cryptocurrency/blockchain), is a fascinating subject; perhaps some readers will look at the vulnerability examples at the following links for inspiration and embark on a journey to discover some more vulnerabilities.

An example of a recently resolved critical problem in Bitcoin, which remained undiscovered for quite some time, can be found at <https://Bitcoincore.org/en/2019/11/08/CVE-2017-18350/>.

Another example of a critical inflation and denial-of-service bug, which was discovered on September 17, 2018 and fixed rapidly, is detailed at <https://Bitcoincore.org/en/2018/09/20/notice/>.

Perhaps there are still some bugs that are yet to be discovered!

Let's now look at the Bitcoin blockchain, which is the foundation of the Bitcoin cryptocurrency.

Blockchain

The Bitcoin blockchain can be defined as a public, distributed ledger holding a timestamped, ordered, and immutable record of all transactions on the Bitcoin network. Transactions are picked up by miners and bundled into blocks for mining. Each block is identified by a hash and is linked to its previous block by referencing the previous block's hash in its header.

Structure

The data structure of a Bitcoin block is shown in the following table:

| Field | Size | Description |
|---------------------|----------|---|
| Block size | 4 bytes | The size of the block. |
| Block header | 80 bytes | This includes fields from the block header described in the next section. |
| Transaction counter | Variable | The field contains the total number of transactions in the block, including the coinbase transaction. The size ranges from 1-9 bytes. |
| Transactions | Variable | All transactions in the block. |

The block header mentioned in the previous table is a data structure that contains several fields. This is shown in the following table:

| Field | Size | Description |
|------------------------------|----------|--|
| Version | 4 bytes | The block version number that dictates the block validation rules to follow. |
| Previous block's header hash | 32 bytes | This is a double SHA-256 hash of the previous block's header. |
| Merkle root hash | 32 bytes | This is a double SHA-256 hash of the Merkle tree of all transactions included in the block. |
| Timestamp | 4 bytes | This field contains the approximate creation time of the block in the Unix epoch time format. More precisely, this is the time when the miner started hashing the header (the time from the miner's location). |
| Difficulty target | 4 bytes | This is the current difficulty target of the network/block. |
| Nonce | 4 bytes | This is a number that miners change repeatedly to produce a hash that is lower than the difficulty target. |

As shown in the following diagram, a blockchain is a chain of blocks where each block is linked to the previous block by referencing the previous block header's hash. This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified. The first block is not linked to any previous block and is known as the genesis block:

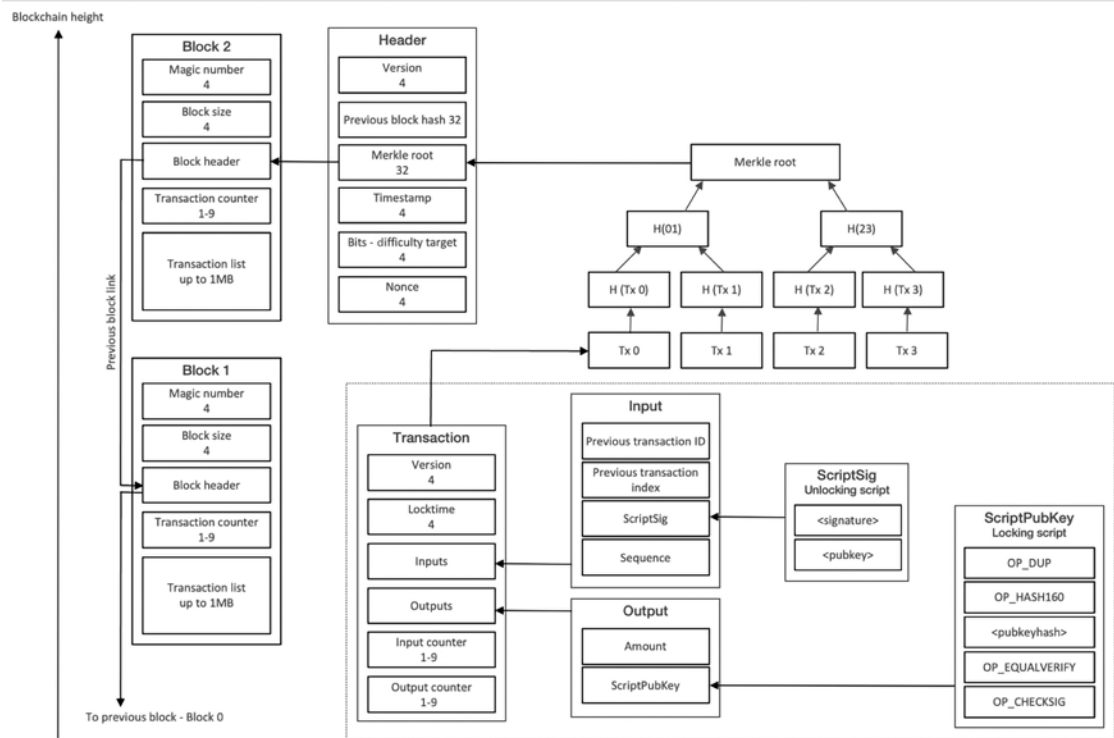


Figure 6.5: A visualization of a blockchain, block, block header, transactions, and scripts

On the left-hand side, blocks are shown starting from bottom to top. Each block contains transactions and block headers, which are further magnified on the right-hand side. At the top, first, the block header is enlarged to show various elements within the block header. Then on the right-hand side, the Merkle root element of the block header is shown in a magnified view, which shows how the Merkle root is constructed.



We have discussed Merkle trees in detail previously. You can refer to *Chapter 4, Asymmetric Cryptography*, if you need to revise the concept.

Further down the diagram, transactions are also magnified to show the structure of a transaction and the elements that it contains. Also, note that transactions are then further elaborated to show what locking and unlocking scripts look like. The size (in bytes) of each field of block, header, and transaction is also shown as a number under the name of the field.

Let's move on to discuss the first block in the Bitcoin blockchain: the genesis block.

The genesis block

This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the Bitcoin core software. In the genesis block, the coinbase transaction included a comment taken from *The Times* newspaper, proving that the Bitcoin genesis block was not mined earlier than January 3, 2009:

"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"

The following representation of the genesis block code can be found in the `chainparams.cpp` file available at <https://github.com/Bitcoin/Bitcoin/blob/master/src/chainparams.cpp>:

```
static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t
nBits, int32_t nVersion, const CAmount& genesisReward)
{
    const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of
second bailout for banks";

    const CScript genesisOutputScript = CScript() <<
ParseHex("04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61
deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f") << OP_
CHECKSIG;

    return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce,
nBits, nVersion, genesisReward);
}
```

The block height is the number of blocks before a particular block in the blockchain. PoW is used to secure the blockchain. Each block contains one or more transactions, the first of which is the coinbase transaction. There is a special condition for coinbase transactions that prevents them from being spent until at least 100 blocks have passed to avoid a situation where the block may be declared stale later.

Stale and orphan blocks

Stale blocks are old blocks that have already been mined. Miners who keep working on these blocks due to a fork, where the longest chain (main chain) has already progressed beyond those blocks, are said to be working on a stale block. In other words, these blocks exist on a shorter chain, and will not provide any reward to their miners.

Orphan blocks are a slightly different concept. Their parent blocks are unknown. As their parents are unknown, they cannot be validated. This problem occurs when two or more miners discover a block at almost the same time. These are valid blocks and were correctly discovered at some point in the past but now they are no longer part of the main chain. The reason why this occurs is that if there are two blocks discovered at almost the same time, the one with a larger amount of **Proof of Work (PoW)** will be accepted and the one with a lower amount of work will be rejected. Similar to stale blocks, they do not provide any reward to their miners.

We can see this concept visually in the following diagram:

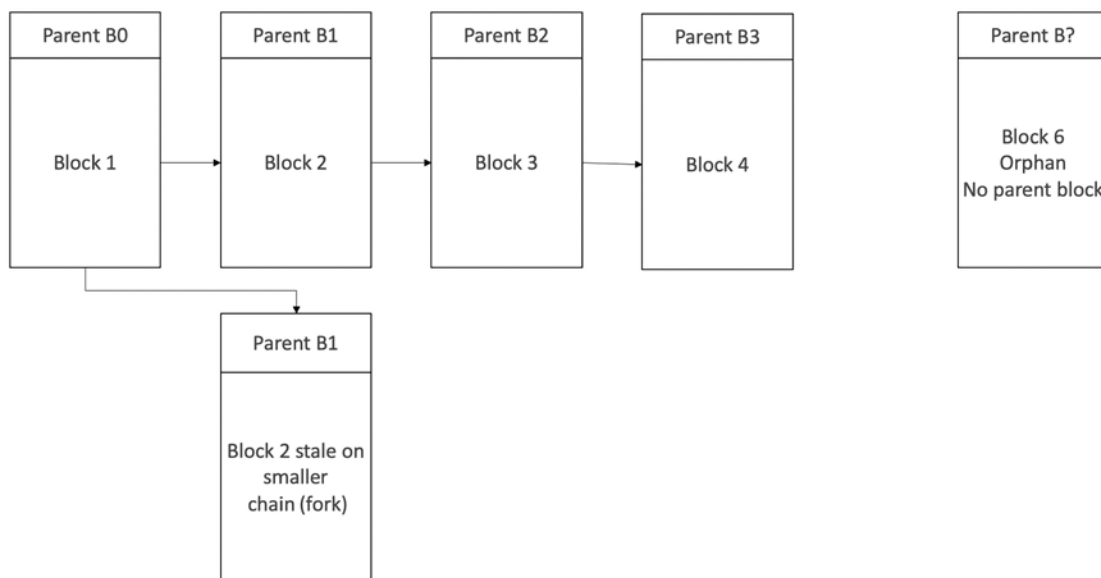


Figure 6.6: Orphan and stale blocks

Forks

In the preceding introduction to stale blocks, we introduced a new term, a **fork**. A fork is a condition that occurs when two different versions of the blockchain exist. It is acceptable in some conditions and detrimental in a few others.

Because of the distributed nature of Bitcoin, network forks can occur inherently. In cases where two nodes simultaneously announce a valid block, it can result in a situation where there are two blockchains with different transactions. This is an undesirable situation but can be addressed by the Bitcoin network only by accepting the longest chain.

In this case, the smaller chain will be considered orphaned. If an adversary manages to gain control of 51% of the network hash rate (computational power), then they can impose their own version of the transaction history.

There are different types of forks that can occur in a blockchain:

- Temporary forks
- Soft forks
- Hard forks

Temporary forks can occur naturally in Bitcoin protocol when two blocks are mined roughly at the same time and a tree-like structure starts to emerge with two or more branches. However, due to the chain choice rule, which enforces the selection of the longest chain among all the branches, the shorter chain is removed. Longest chain means the chain that has the most accumulated work (difficulty) behind it.

In the case of a **soft fork**, a client that chooses not to upgrade to the latest version supporting the updated protocol will still be able to work and operate normally. In this case, new and previous blocks are both acceptable, thus making a soft fork backward compatible. Miners are only required to upgrade to the new soft-fork client software to make use of the new protocol rules. Planned upgrades do not necessarily create forks because all users should have updated the software already.

A **hard fork**, on the other hand, invalidates previously valid blocks and requires all users to upgrade. New transaction types are sometimes added as a soft fork, and any changes such as block structure changes or major protocol changes result in a hard fork.

As Bitcoin evolves and new upgrades and innovations are introduced in it, the version associated with blocks also changes. These versions introduce various security parameters and new features.



Details of BIP0065 are available at <https://github.com/Bitcoin/bips/blob/master/bip-0065.mediawiki>.

Properties

Bitcoin is an ever-growing chain of blocks and is increasing in size. The current size of the Bitcoin blockchain stands at approximately 432 GB. The graph at <https://www.blockchain.com/charts/blocks-size> shows the current and historic size. As the chain grows and more miners are added to the network, the network difficulty also increases. Network difficulty refers to a measure of how difficult it is to find a new block, or in other words, how difficult it is to find a hash below the given target.

New blocks are added to the blockchain approximately every 10 minutes, and the network difficulty is adjusted dynamically every 2,016 blocks (roughly every two weeks) in order to maintain a steady addition of new blocks to the network.

Network difficulty is calculated using the following formula:

$$\text{Difficulty}_{\text{new}} = \frac{\text{Difficulty}_{\text{previous}} \times (2016 \times 10 \text{ minutes})}{\text{time taken to mine last 2016 blocks}}$$

The *previous difficulty* represents the old target value, and 2016×10 is the *total* time taken to generate the previous 2,016 blocks. Network difficulty essentially means how hard it is for miners to find a new block; that is, how difficult the hashing puzzle is now.

In the next section, mining is discussed, which will explain how the hashing puzzle is solved.

Miners

Mining is a process by which new blocks are added to the blockchain. This process is resource-intensive due to the requirements of PoW, where miners compete to find a number less than the difficulty target of the network. This difficulty in finding the correct value (also sometimes called the **mathematical puzzle**) is there to ensure that miners have spent the required resources before a new proposed block can be accepted. The miners mint new coins by solving the PoW problem, also known as the partial hash inversion problem. This process consumes a high amount of resources, including computing power and electricity. This process also secures the system against fraud and double-spending attacks while adding more virtual currency to the Bitcoin ecosystem.

Roughly one new block is created (mined) every 10 minutes to control the frequency of the generation of bitcoins. This frequency needs to be maintained by the Bitcoin network. It is encoded in the Bitcoin Core client to control the “money supply.”

Approximately 144 blocks, that is, 1,728 bitcoins, are generated per day. The number of actual coins can vary per day; however, the number of blocks remains at an average of 144 per day. Bitcoin supply is also limited. In 2140, all 21 million bitcoins will have finally been created, and no new bitcoins will be able to be created after that. Bitcoin miners, however, will still be able to profit from the ecosystem by charging transaction fees.

Once a node connects to the Bitcoin network, there are several tasks that a Bitcoin miner performs:

1. **Syncing up with the network:** Once a new node joins the Bitcoin network, it downloads the blockchain by requesting historical blocks from other nodes. This is mentioned here in the context of the Bitcoin miner; however, this is not necessarily a task that only concerns miners. Other nodes, such as full nodes who are not necessarily mining, also sync with the network to update the local blockchain database.
2. **Transaction validation:** Transactions broadcast on the network are validated by full nodes by verifying and validating signatures and outputs.
3. **Block validation:** Miners and full nodes can start validating blocks received by them by evaluating them against certain rules. This includes the verification of each transaction in the block along with the verification of the nonce value.
4. **Creating a new block:** Miners propose a new block by combining transactions broadcast on the network after validating them.

5. **Performing PoW:** This task is the core of the mining process, and this is where miners find a valid block by solving a computational puzzle. The block header contains a 32-bit nonce field and miners are required to repeatedly vary the nonce until the resultant hash is less than a predetermined target.
6. **Fetch reward:** Once a node solves the hash puzzle (PoW), it immediately broadcasts the results, and other nodes verify it and accept the block. There is a slight chance that the newly minted block will not be accepted by other miners on the network due to a clash with another block found at roughly the same time, but once accepted, the miner is rewarded with bitcoins and any associated transaction fees.

Miners are rewarded with new coins if and when they discover new blocks by solving the PoW. Miners are paid transaction fees in return for the transactions in their proposed blocks. New blocks are created at an approximate fixed rate of every 10 minutes.

The rate of creation of new bitcoins decreases by 50% every 210,000 blocks, which is roughly every 4 years. When Bitcoin started in 2009, the mining reward used to be 50 bitcoins. After every 210,000 blocks, the block reward halves. In November 2012, it halved down to 25 bitcoins. Currently, since May 2020, it is 6.25 bitcoins per block. The next halving is expected to occur in early 2024, which will reduce the reward to 3.125 BTC. This mechanism is hardcoded in Bitcoin to regulate and control inflation and limit the supply of bitcoins. For miners to earn the reward, they must show that they have solved the computational puzzle. This is called the PoW.

Proof of Work (PoW)

This is proof that enough computational resources have been spent to build a valid block. PoW is based on the idea that a random node is selected every time to create a new block. In this model, nodes compete in proportion to their computing capacity to be selected. The following formula sums up the PoW requirement in Bitcoin:

$$H(N || P_hash || Tx || Tx || \dots Tx) < Target$$

Here, N is a nonce, P_hash is a hash of the previous block, Tx represents the transactions in the block, and $Target$ is the target network difficulty value. This means that the hash of the previously mentioned concatenated fields should be less than the target hash value.

The only way to find this nonce is the brute force method. Once a certain pattern of a certain number of zeroes is met by a miner, the block is immediately broadcast and accepted by other miners.

The mining algorithm consists of the following steps:

1. The previous block's header is retrieved from the Bitcoin network.
2. Assemble a set of transactions broadcast on the network into a block to be proposed.
3. Compute the double hash of the previous block's header, combined with a nonce and the newly proposed block, using the SHA-256 algorithm.
4. Check if the resulting hash is lower than the current difficulty level (the target). If so, then the PoW is solved. As a result of successful PoW, the discovered block is broadcast to the network and miners fetch the reward.

5. If the resultant hash is not less than the current difficulty level (target), then repeat the process after incrementing the nonce.

As the hash rate of the Bitcoin network increased, the total amount of the 32-bit nonce was exhausted too quickly. To address this issue, the extra nonce solution was implemented, whereby the coinbase transaction is used to provide a larger range of nonces to be searched by the miners. This process is visualized in the following flowchart:

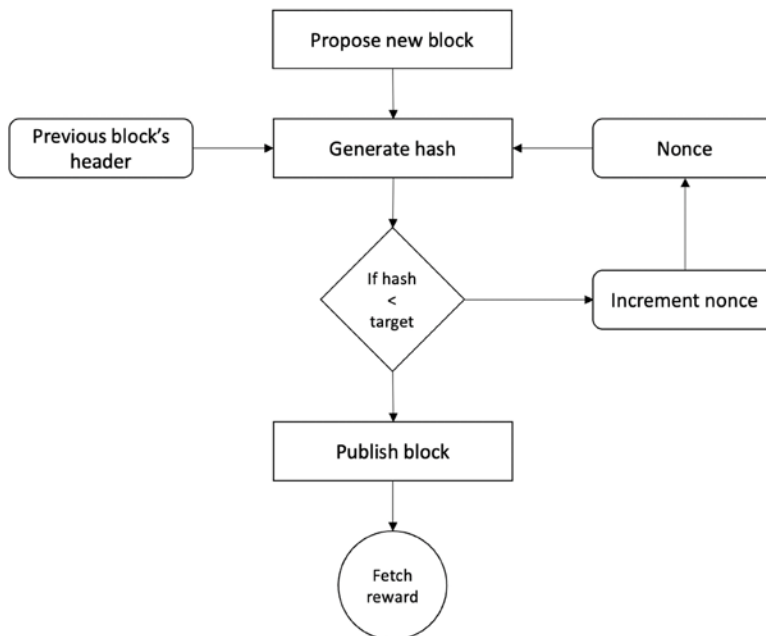


Figure 6.7: Mining process

Mining difficulty increases over time and bitcoins that could once be mined by a single-CPU laptop computer now require dedicated mining centers to solve the hash puzzle. The current difficulty level can be queried through the Bitcoin command-line interface using the following command:

```
$ bitcoin-cli getdifficulty
```

This generates something like the following:

```
36,835,682,546,787.98
```

This number represents the PoW difficulty level of the Bitcoin network. Recall from previous sections that miners compete to find a solution to a problem. This number, in fact, shows how difficult it is to find a hash lower than the network difficulty target. All successfully mined blocks must contain a hash that is less than this target number. This number is updated every 2 weeks or 2,016 blocks to ensure that on average, the 10-minute block generation time is maintained. Bitcoin network difficulty has increased in a roughly exponential fashion.



The difficulty of the Bitcoin network has increased quite significantly over the last few years. The latest difficulty graph is available here: <https://www.blockchain.com/charts/difficulty>.

The reason why mining difficulty increases is because, in Bitcoin, the block generation time always must be around ten minutes. This means that if blocks are being mined too quickly because of faster hardware, the difficulty increases accordingly so that the block generation time can remain at roughly ten minutes per block. This phenomenon is also true in reverse. If blocks take longer than ten minutes on average to mine, then the difficulty is decreased. The difficulty is calculated every 2,016 blocks (around two weeks) and adjusted accordingly. If the previous set of 2,016 blocks were mined in a period of less than two weeks, then the difficulty increases. Similarly, if 2,016 blocks were found in more than two weeks (bearing in mind that if blocks are mined every ten minutes, then 2,016 blocks take two weeks to be mined), then the difficulty is decreased.

The Bitcoin miners calculate hashes to solve the PoW algorithm. If the difficulty goes up then a higher hash rate is required to find the blocks. The difficulty increases accordingly if more hashing power is added due to more miners joining the network. The hash rate basically represents the rate of hash calculation per second. In other words, this is the speed at which miners in the Bitcoin network are calculating hashes to find a block.



The hash rate increases over time and is currently at 266 EH. To see the latest figure, a graph is available here: <https://www.blockchain.com/charts/hash-rate>

Mining systems

Over time, Bitcoin miners have used various methods to mine bitcoins. As the core principle behind mining is based on the double SHA-256 algorithm, over time, experts have developed sophisticated systems to calculate the hash faster and faster. The following is a review of the different types of mining methods used in Bitcoin and how they have evolved with time.

CPU

CPU mining was the first type of mining available in the original Bitcoin client. Users could even use laptop or desktop computers to mine bitcoins. CPU mining is no longer profitable and now more advanced mining methods such as ASIC-based mining are used. CPU mining only lasted for around a year from the introduction of Bitcoin, and soon other methods were explored and tried by miners.

GPU

Due to the increased difficulty of the Bitcoin network and the general tendency of finding faster methods to mine, miners started to use the GPUs or graphics cards available in PCs to perform mining. GPUs support faster and parallelized calculations, which are usually programmed using the OpenCL language.

This turned out to be a faster option as compared to CPUs. Users also used techniques such as over-clocking to gain the maximum benefit of the GPU power. Also, the possibility of using multiple graphics cards in parallel increased the popularity of graphics cards' usage for Bitcoin mining. GPU mining, however, has some limitations, such as overheating and the requirement for specialized motherboards and extra hardware to house multiple graphics cards. From another angle, graphics cards have become quite expensive due to increased demand, and this has impacted gamers and graphics software users.

FPGAs

Even GPU mining did not last long, and soon miners found another way to perform mining using **Field Programmable Gate Arrays (FPGAs)**.



It should be noted that GPU mining is still profitable for some other cryptocurrencies to some extent, because the network difficulty is much lower than that of Bitcoin.

An FPGA is basically an integrated circuit that can be programmed to perform specific operations. FPGAs are usually programmed in **Hardware Description Languages (HDLs)**, such as Verilog and VHDL. Double SHA-256 quickly became an attractive programming task for FPGA programmers and several open-source projects were started too. FPGAs offered much better performance compared to GPUs; however, issues such as accessibility, programming difficulty, and the requirement for specialized knowledge to program and configure FPGAs resulted in a short life for the FPGA era of Bitcoin mining.

Mining hardware such as the X6500 miner, Ztex, and Icarus was developed during the time when FPGA mining was profitable. Various FPGA manufacturers, such as Xilinx and Altera, produce FPGA hardware and development boards that can be used to program mining algorithms.

ASICs

ASICs were designed to perform SHA-256 operations. These special chips were sold by various manufacturers and offered a very high hashing rate. The arrival of ASICs resulted in the quick phasing out of FPGA-based systems for mining.

This worked for some time, but due to the quickly increasing mining difficulty level, single-unit ASICs are no longer profitable. With the current difficulty factor (as of May 2022), if a miner manages to produce a hash rate of 100 trillion hashes per second (TH/s), they can hope to make 0.0004374 BTC (around \$12) per day, and around \$4,456.61 a year, which is very low compared to the investment required to source the equipment that can produce 100 TH/s, roughly 20,000 USD. Including running costs such as electricity, this turns out to be not very profitable at all.

Now, professional mining centers using thousands of ASIC units in parallel are offering mining contracts to users to perform mining on their behalf. There is no technical limitation—a single user can run thousands of ASICs in parallel—but it will require dedicated data centers and hardware; therefore, the cost for an individual can become prohibitive.

Mining pools

A mining pool forms when a group of miners work together to mine a block. The pool manager receives the coinbase transaction if the block is successfully mined and is then responsible for distributing the reward to the group of miners who invested resources to mine the block. This is more profitable than solo mining, where only one sole miner is trying to solve the partial hash inversion function (hash puzzle) because, in mining pools, the reward is paid to each member of the pool regardless of whether they (or more specifically, their individual node) solved the puzzle or not.

There are various models that a mining pool manager can use to pay miners, such as the pay-per-share model and the proportional model. In the pay-per-share model, the mining pool manager pays a flat fee to all miners who participated in the mining exercise, whereas in the proportional model, the share is calculated based on the amount of computing resources spent to solve the hash puzzle.



Many commercial pools now exist and provide mining service contracts via the cloud and easy-to-use web interfaces. The most used ones are AntPool, ViaBTC, F2Pool, and Poolin. A comparison of the hashing power of major mining pools is available here: <https://blockchain.info/pools>

Mining centralization can occur if a pool manages to control more than 51% of the network by generating more than 51% of the hash rate of the Bitcoin network. As discussed earlier in the introduction, a 51% attack can result in successful double-spending attacks, and it can impact consensus and in fact, even impose another version of the transaction history on the Bitcoin network. This event has happened once in Bitcoin history when GHash.io, a large mining pool, managed to acquire more than 51% of the network capacity. Theoretical solutions, such as two-phase PoW, have been proposed in academia to disincentivize large mining pools.



The following article describes a two-phase PoW proposal, in order to disincentivize large Bitcoin mining pools: <http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/>.

This scheme introduces a second cryptographic puzzle that results in mining pools either revealing their private keys or providing a considerable portion of the hash rate of their mining pool, thus reducing the overall hash rate of the pool.

The race to build efficient miners is on and is only expected to grow, though not infinitely. However, there is a limit to hardware acceleration and physical limitations. Soon, no room will be left for any on-chip optimizations. Or, it may become extremely challenging to achieve any further optimizations unless some fundamental change occurs in the way hardware silicone is developed.

Network

The Bitcoin network is a **P2P** network where nodes perform transactions. They verify and propagate transactions and blocks. As we have just seen, nodes called miners also produce blocks. A full Bitcoin node performs four functions. These are wallet, miner, blockchain, and network routing.

A Bitcoin network is identified by its magic value. Magic values are used to indicate the message's origin network. A list of these networks and values is shown in the following table:

| Network | Magic value | Description |
|-----------------|-------------|--|
| main | 0xD9B4BEF9 | Bitcoin main network |
| testnet | 0xDAB5BFFA | First Bitcoin test network |
| testnet3 | 0x0709110B | Current Bitcoin test network |
| signet(default) | 0x40CF030A | Bitcoin test network with an additional signature requirement for block validation |

Before we examine how the Bitcoin discovery protocol and block synchronization work, we need to understand the different types of messages that the Bitcoin protocol uses.

Types of messages

There are 27 types of protocol messages in total, but that's likely to increase over time as the protocol grows. The most used protocol messages and an explanation of them are listed as follows:

- **Version:** This is the first message that a node sends out to the network, advertising its version and block count. The remote node then replies with the same information and the connection is then established.
- **Verack:** This is the response of the version message accepting the connection request.
- **Inv:** This is used by nodes to advertise their knowledge of blocks and transactions.
- **Getdata:** This is a response to inv, requesting a single block or transaction identified by its hash.
- **Getblocks:** This returns an inv packet containing the list of all blocks starting after the last known hash or 500 blocks.
- **Getheaders:** This is used to request block headers in a specified range.
- **Tx:** This is used to send a transaction as a response to the getdata protocol message.
- **Block:** This sends a block in response to the getdata protocol message.
- **Headers:** This packet returns up to 2,000 block headers as a reply to the getheaders request.
- **Getaddr:** This is sent as a request to get information about known peers.
- **Addr:** This provides information about nodes on the network. It contains the number of addresses and an address list in the form of an IP address and port number.
- **Ping:** This message is used to confirm if the TCP/IP network connection is active.
- **Pong:** This message is the response to a ping message confirming that the network connection is live.

When a Bitcoin Core node starts up, first, it initiates the discovery of all peers. This is achieved by querying DNS seeds that are hardcoded into the Bitcoin Core client and are maintained by Bitcoin community members. This lookup returns several DNS A records. The Bitcoin protocol works on TCP port 8333 by default for the main network and TCP 18333 for the testnet.



DNS seeds are declared (hardcoded) in the `chainparams.cpp` file in the Bitcoin source code, which can be viewed on GitHub at the following link: <https://github.com/bitcoin/bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/chainparams.cpp#L116>.

First, the client sends a protocol message, `version`, which contains various fields, such as the version, services, timestamp, network address, nonce, and some other fields. The remote node responds with its own `version` message, followed by a `verack` message exchange between both nodes, indicating that the connection has been established.

After this, `getaddr` and `addr` messages are exchanged to find the peers that the client does not know. Meanwhile, either of the nodes can send a `ping` message to see whether the connection is still active. `getaddr` and `addr` are message types defined in the Bitcoin protocol.

This process is shown in the following diagram of the protocol:

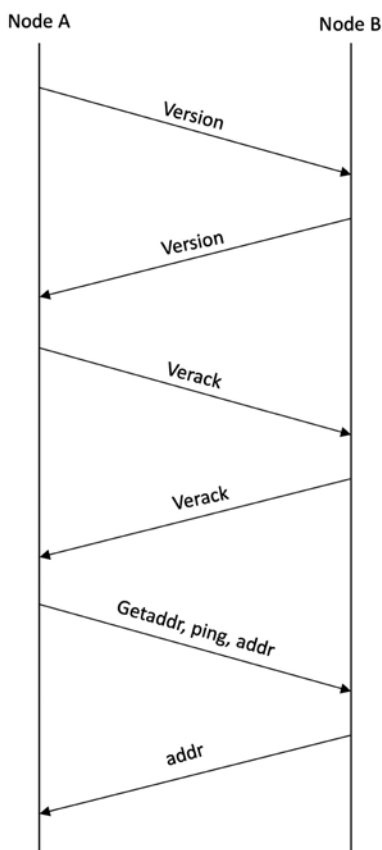


Figure 6.8: Visualization of node discovery protocol

This network protocol sequence diagram shows communication between two Bitcoin nodes during initial connectivity. **Node A** is shown on the left-hand side and **Node B** on the right.

First, **Node A** starts the connection by sending a version message that contains the version number and current time to the remote peer, **Node B**. **Node B** then responds with its own version message containing the version number and current time. **Node A** and **Node B** then exchange a verack message, indicating that the connection has been successfully established. Once this connection is successful, the peers can exchange getaddr and addr messages to discover other peers on the network.

Now, the block download can begin. In version 0.10.0, the initial block download method named *headers-first* was introduced. This resulted in major performance improvement, and blockchain synchronization that used to take days to complete started taking only a few hours. The core idea is that the new node first asks peers for block headers and then validates them. Once this has been completed, blocks are requested in parallel from all available peers. This happens because the blueprint of the complete chain is already downloaded in the form of the block header chain.

In this method, when the client starts up, it checks whether the blockchain is fully synchronized if the header chain is already synchronized; if not, which is the case the first time the client starts up, it requests headers from other peers using the getheaders message. If the blockchain is fully synchronized, it listens for new blocks via inv messages, and if it already has a fully synchronized header chain, then it requests blocks using getdata protocol messages. The node also checks whether the header chain has more headers than blocks, and then it requests blocks by issuing the getdata protocol message:

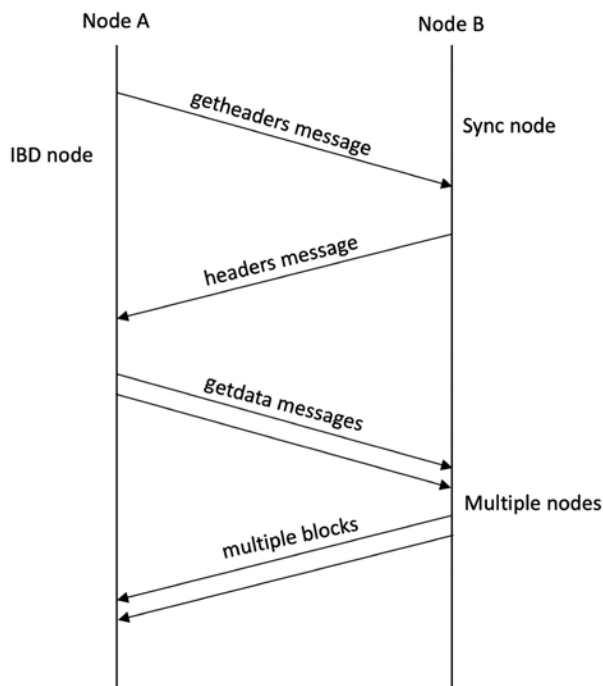



Figure 6.9: Bitcoin Core client $\geq 0.10.0$ header and block synchronization

The preceding diagram shows the Bitcoin block synchronization process between two nodes on the Bitcoin network. **Node A**, shown on the left-hand side, is called an **Initial Block Download (IBD)** node, and **Node B**, shown on the right, is called a **sync node**.

A protocol graph showing the flow of data between the two peers can be seen in the preceding screenshot. This can help you understand when a node starts up and what type of messages are used. This view can be accessed by selecting the appropriate network card interface and then applying the filter for Bitcoin.

In the following example, the Bitcoin dissector is used to analyze the traffic and identify the Bitcoin protocol commands. The exchange of messages such as version, getaddr, and getdata can be seen, along with the appropriate comment describing the message name.

This can be a very useful exercise in order to learn about the Bitcoin protocol and it is recommended that experiments be carried out on the Bitcoin testnet (<https://en.bitcoin.it/wiki/Testnet>), where various messages and transactions can be sent over the network and then analyzed by Wireshark.



Wireshark is a network analysis tool and is available at <https://www.wireshark.org>.

The analysis being performed here by Wireshark shows messages being exchanged between two nodes. If you look closely, you'll notice that the top three messages show the node discovery protocol that we introduced earlier:

| Time | 192.168.0.13 | 136.243.139.96 | Comment |
|---------------|--------------|------------------------|--|
| 97.734135000 | (57868) | version → (18333) | Bitcoin: version |
| 98.025045000 | (57868) | verack → (18333) | Bitcoin: verack |
| 98.025177000 | (57868) | getaddr.ping → (18333) | Bitcoin: getaddr, ping, addr |
| 98.025468000 | (57868) | getheaders → (18333) | Bitcoin: getheaders, [unknown command], [unknown command], [unknown command], headers |
| 98.160419000 | (57868) | [TCP Retran → (18333) | Bitcoin: [TCP Retransmission], getheaders, [unknown command], [unknown command], [unknown command] |
| 98.598399000 | (57868) | getdata → (18333) | Bitcoin: getdata |
| 144.343544000 | (57868) | inv → (18333) | Bitcoin: inv |
| 176.152240000 | (57868) | getdata → (18333) | Bitcoin: getdata |
| 179.493755000 | (57868) | getdata → (18333) | Bitcoin: getdata |
| 218.101646000 | (57868) | ping → (18333) | Bitcoin: ping |
| 218.192004000 | (57868) | [unknown.co → (18333) | Bitcoin: [unknown command] |
| 218.444431000 | (57868) | [TCP Retran → (18333) | Bitcoin: [TCP Retransmission], [unknown command] |
| 336.234936000 | (57868) | getdata → (18333) | Bitcoin: getdata |
| 337.843423000 | (57868) | [unknown.co → (18333) | Bitcoin: [unknown command] |
| 338.143885000 | (57868) | ping → (18333) | Bitcoin: ping |
| 448.764093000 | (57868) | getdata → (18333) | Bitcoin: getdata |
| 457.894823000 | (57868) | [unknown.co → (18333) | Bitcoin: [unknown command] |
| 458.195265000 | (57868) | ping → (18333) | Bitcoin: ping |
| 578.011774000 | (57868) | [unknown.co → (18333) | Bitcoin: [unknown command] |
| 578.212044000 | (57868) | ping → (18333) | Bitcoin: ping |
| 585.587671000 | (57868) | inv → (18333) | Bitcoin: inv |
| 647.169633000 | (57868) | inv → (18333) | Bitcoin: inv |
| 671.962545000 | (57868) | getdata → (18333) | Bitcoin: getdata |
| 698.037067000 | (57868) | [unknown.co → (18333) | Bitcoin: [unknown command] |
| 698.237350000 | (57868) | ping → (18333) | Bitcoin: ping |
| 701.563581000 | (57868) | inv → (18333) | Bitcoin: inv |
| 701.986269000 | (57868) | inv → (18333) | Bitcoin: inv |
| 705.022173000 | (57868) | inv → (18333) | Bitcoin: inv |
| 812.115878000 | (57868) | inv → (18333) | Bitcoin: inv |
| 818.198570000 | (57868) | [unknown.co → (18333) | Bitcoin: [unknown command] |
| 818.298733000 | (57868) | ping → (18333) | Bitcoin: ping |

Figure 6.11: Bitcoin node discovery protocol in Wireshark

Nodes run different Bitcoin client software. We'll discuss those next.

Client software

There are different types of nodes on the network. The two main types of nodes are full nodes and **simple payment verification (SPV)** nodes.

Full nodes, as the name implies, are implementations of Bitcoin Core clients performing the wallet, miner, full blockchain storage, and network routing functions. These nodes download the entire blockchain; they provide the most secure method of validating the blockchain as a client and play a vital role in block propagation. However, it is not necessary for all nodes in a Bitcoin network to perform all these functions. Some nodes perform network routing functions only but do not perform mining or store private keys (the wallet function). Another type of node is solo miner nodes, which can perform mining, store full blockchains, and act as Bitcoin network routing nodes. Some nodes perform only mining functions and are called mining nodes.



Versioning information is coded in the Bitcoin client in the `version.h` file, which is available here: <https://github.com/bitcoin/bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/version.h#L9>.

It is possible to run SPV software that runs a wallet and network routing function without a blockchain while syncing with the network. SPV nodes only keep a copy of block headers of the current longest valid blockchain. When required, they can request transactions from full nodes. Verification is performed by looking at the Merkle branch, which links the transactions to the original block the transaction was accepted in. This is not very practical and requires a more pragmatic approach, which was implemented with BIP37, where bloom filters were used to filter for relevant transactions only. We will review bloom filters in the next section.

Finally, there are a few nonstandard but heavily used nodes. These are called pool protocol servers. These nodes make use of alternative protocols such as the stratum protocol, a line-based protocol that makes use of plain TCP sockets and human-readable JSON-RPC to operate and communicate between nodes. Stratum is commonly used to connect to mining pools. Nodes that only compute hashes use the Stratum protocol to submit their solutions to the mining pool.



Most protocols on the internet are line-based, which means that each line is delimited by a carriage return and newline `\r \n` character. More details on this protocol are available at this link: https://en.bitcoin.it/wiki/Stratum_mining_protocol.

Bloom filters

A bloom filter is a data structure (a bit vector with indexes) that is used to test the membership of an element in a probabilistic manner. It provides a probabilistic lookup with false positives but no false negatives.

This means that this filter can produce an output where an element that is not a member of the set being tested is wrongly considered to be in the set. Still, it can never produce an output where an element does exist in the set, but it asserts that it does not.

Elements are added to the bloom filter after hashing them several times and then setting the corresponding bits in the bit vector to 1 via the corresponding index. To check the presence of an element in the bloom filter, the same hash functions are applied and then compared with the bits in the bit vector to see whether the same bits are set to 1.

Note that not every hash function (such as SHA-1) is suitable for bloom filters as they need to be fast, independent, and uniformly distributed. Noncryptographic hash functions are used for bloom filters such as fnv, murmur, and Jenkins.

These filters are mainly used by SPV clients to request transactions and the Merkle blocks that they are interested in. A Merkle block is a lightweight version of the block, which includes a block header, some hashes, a list of 1-bit flags, and a transaction count. This information can then be used to build a Merkle tree. This is achieved by creating a filter that matches only those transactions and blocks from the full chain that have been requested by the SPV client. Once version messages have been exchanged and the connection is established between the peers, the nodes can set filters according to their requirements.

These probabilistic filters offer a varying degree of privacy or precision, depending on how accurately or loosely they have been set. A strict bloom filter will only filter transactions that have been requested by the node, but at the expense of the possibility of revealing the user addresses to adversaries who can correlate transactions with their IP addresses, thus compromising privacy.

On the other hand, a loosely set filter can result in retrieving more unrelated transactions but will offer more privacy. Also, for SPV clients, bloom filters allow them to use low bandwidth as opposed to downloading all transactions for verification.

BIP37 proposed the Bitcoin implementation of bloom filters and introduced three new messages to the Bitcoin protocol:

- **filterload:** This is used to set the bloom filter on the connection.
- **filteradd:** This adds a new data element to the current filter.
- **filterclear:** This deletes the currently loaded filter.



More details can be found in the BIP37 specification. This is available at <https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki>.

So far, we've discussed that, on a Bitcoin network, there are full clients (nodes), which perform the function of storing a complete blockchain. If you cannot run a full node, then SPV clients can be used to verify that particular transactions are present in a block by only downloading the block headers instead of the entire blockchain.

At times, even running an SPV node is not feasible (especially on low-resource devices such as mobile phones) and the requirement is only to be able to send and receive Bitcoin somehow. For this purpose, wallets (wallet software) are used that do not require downloading even the block headers.

Wallets

The wallet software is used to generate and store cryptographic keys. It performs various useful functions, such as receiving and sending Bitcoin, backing up keys, and keeping track of the balance available. Bitcoin client software usually offers both functionalities: a Bitcoin client and wallet. On disk, the Bitcoin Core client wallets are stored as a Berkeley DB file:

```
$ file wallet.dat
wallet.dat: Berkeley DB (Btree, version 9, native byte-order)
```

Private keys are generated by randomly choosing a 256-bit number provided by the wallet software. The rules of generation are predefined and were discussed in *Chapter 4, Asymmetric Cryptography*. Private keys are used by wallets to sign outgoing transactions. Wallets do not store any coins. In fact, in the Bitcoin network, coins do not exist; instead, only transaction information is stored on the blockchain (more precisely, UTXO, unspent outputs), which are then used to calculate the number of bitcoins.

Fundamentally, a cryptocurrency wallet depends on a **keystore** to store private keys. A keystore can be defined as a repository for storing keys. It can be as simple as a file or as complex as a **hardware security module (HSM)** or even a portable device such as a hardware wallet. The private key(s) must be protected against theft. Usually, encryption of the keystore and specialized secure hardware such as an HSM are used to protect the keys. These private keys are used in digital signatures to sign the transactions as proof of the ownership of the coins.

In Bitcoin and generally in cryptocurrencies, there are different types of wallets that can be used to store private keys. As software, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network. Let's look at the common types of wallets:

- **Non-deterministic wallets:** These wallets contain randomly generated private keys and are also called **Just a Bunch of Key** wallets. The Bitcoin Core client generates some keys when first started and also generates keys as and when required. Managing many keys is very difficult and an error-prone process that can lead to the theft and, consequently, the loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately, for example, by encrypting them to prevent theft or loss.
- **Deterministic wallets:** In this type of wallet, keys are derived from a seed value via hash functions. This seed number is generated randomly and is commonly represented by human-readable **mnemonic code** words. Mnemonic code words are defined in BIP39, a Bitcoin improvement proposal for mnemonic code for generating deterministic keys. This BIP is available at <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki>. These phrases can be used to recover all keys and make private key management comparatively easier.
- **Hierarchical deterministic (HD) wallets:** Defined in BIP32 and BIP44, HD wallets store keys in a tree structure derived from a seed.

The seed generates the parent key (master key), which is used to generate child keys and, subsequently, grandchild keys. Key generation in HD wallets does not generate keys directly; instead, it produces some information (private key generation information) that can be used to generate a sequence of private keys.

The complete hierarchy of private keys in an HD wallet is easily recoverable if the master private key is known. It is because of this property that HD wallets are very easy to maintain and are highly portable. There are many free and commercial HD wallets available, for example, **Trezor** (<https://trezor.io>), **Jaxx** (<https://jaxx.io>), and **Electrum** (<https://electrum.org/>).

- **Brain wallets:** The master private key can also be derived from a single human-readable and easy-to-remember (hence the phrase brain wallets) phrase. The idea here is that this easy-to-memorize passphrase is used to derive the private key and if used in HD wallets, this can result in a full HD wallet that is derived from a single memorized password. This is known as a brain wallet. While this is an easy-to-use solution for users, this method is prone to password guessing, dictionary attacks, and brute-force attacks. Some techniques such as **key stretching** can be used to slow down the progress made by the attacker.
- **Paper wallets:** As the name implies, this is a paper-based wallet with the required key material printed on it. It requires physical security to be stored. Paper wallets can be generated online from various service providers, such as <https://bitcoinpaperwallet.com/> or <https://www.bitaddress.org/>.
- **Hardware wallets:** Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom-built. With the advent of **Near-Field Communication (NFC)** enabled phones, this can also be a **secure element (SE)** in NFC phones. Trezor (<https://trezor.io>) and Ledger wallets (<https://www.ledger.com>) are the commonly used cryptocurrency hardware wallets.



Secure elements tamper-resistant chip that can securely host applications, confidential data, and cryptographic keys. Mobile smartphones, cryptocurrency hardware wallets, and tablets use secure elements to securely store sensitive data such as passwords, PIN codes, fingerprints, and in fact anything requiring confidentiality and integrity.

NFC chips short-range wireless communication between two devices in close proximity and is commonly used for contactless payments and other secure applications through smartphones.

- **Online wallets:** Online wallets, as the name implies, are stored entirely online and are provided as a service, usually via the cloud. They provide a web interface for users to manage their wallets and perform various functions, such as making and receiving payments. They are easy to use but require that the user trusts the online wallet service provider. An example of an online wallet is **GreenAddress**, which is available at <https://greenaddress.it/en/>.
- **Mobile wallets:** Mobile wallets, as the name suggests, are installed on mobile devices. They can provide us with various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments. There are many companies offering these wallets. It is, however, unwise to suggest which type of wallet should be used as it varies depending on personal preferences and the features available in the wallet. Therefore, we will not recommend any here.



An SPV client, which we introduced earlier, is also a type of Bitcoin wallet. Other types of Bitcoin wallets, especially mobile wallets, are mostly API wallets that rely on a mechanism where private and public keys are stored locally on the device where the wallet software is installed. Here, trusted backend servers are used for providing blockchain data via APIs.

The choice of Bitcoin wallet depends on several factors, such as security, ease of use, and available features. Out of all these attributes, security, of course, comes first, and when deciding which wallet to use, security should be of paramount importance.

Hardware wallets tend to be more secure compared to web wallets because of their tamper-resistant design. Web wallets, by their very nature, are hosted on websites, which may not be as secure as a tamper-resistant hardware device. Generally, mobile wallets for smartphone devices are quite popular due to a balanced combination of features, user experience, and security.

Summary

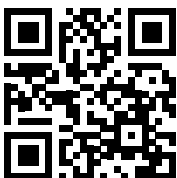
This chapter started with an introduction to Bitcoin and explained how a transaction works from the user's point of view. Then, an introduction to transactions from a technical point of view was presented. After this, the public and private keys used in Bitcoin were discussed. In the next section, we introduced addresses and their different types, followed by a discussion on transactions and their types and usage.

Following this, we looked at blockchain, with a detailed explanation of how blockchain works and the various components included in the Bitcoin blockchain. Next, mining processes and relevant concepts such as hardware systems, their limitations, and bitcoin rewards were introduced, along with an introduction to the Bitcoin network, followed by a discussion on Bitcoin node discovery and block synchronization protocols. Finally, we examined different types of Bitcoin wallets and discussed the various attributes and features of each type.

In the next chapter, we will examine some practical concepts related to Bitcoin payments, clients, and programming.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>